THE RULEIT METHODOLOGY

Olesia Brill, Constanze Deiters, Ursula Goltz, Sandra Lange, Benjamin Mensing, Andreas Rausch, Kurt Schneider

23. December 2010 | NTH Computer Science Report 2010/03

This work was funded by the NTH Focused Research School for IT Ecosystems. NTH (Niedersächsische Technische Hochschule) is a joint university consisting of Technische Universität Braunschweig, Technische Universität Clausthal, and Leibniz Universität Hannover.

IMPRESSUM

Publisher

NTH Focused Research School for IT Ecosystems
Technische Universität Clausthal, Julius-Albert Str. 4, 38678 Clausthal-Zellerfeld, Germany

Editors of the series

Prof. Dr. Christian Müller-Schloer
Prof. Dr. Andreas Rausch
Prof. Dr. Lars Wolf

Technical editor

Dirk Niebuhr
Contact: dirk.niebuhr@tu-clausthal.de

NTH Computer Science Report Review Board

Prof. Dr. Jiří Adámek
Prof. Dr. Jürgen Dix
Prof. Dr. Ursula Goltz
Prof. Dr. Jörg Hähner
Dr. Michaela Huhn
Prof. Dr. Jörg P. Müller
Prof. Dr. Christian Müller-Schloer
Prof. Dr. Wolfgang Nejdl
Dirk Niebuhr
Prof. Dr. Niels Pinkwart
Prof. Dr. Andreas Rausch
Prof. Dr. Kurt Schneider
Prof. Dr. Christian Siemers
Prof. Dr. Heribert Vollmer
Prof. Dr. Mark Vollrath
Prof. Dr.-Ing. Bernardo Wagner
Prof. Dr. Klaus-Peter Wiedmann
Prof. Dr.-Ing. Lars Wolf

ISBN 978-3-942216-15-9

AUTHORS

Dipl.-Math. Olesia Brill	Leibniz Universität Hannover
Dipl.-Inf. Constanze Deiters	TU Clausthal
Prof. Dr. Ursula Goltz	TU Braunschweig
Dipl.-Inf. Sandra Lange	TU Clausthal
B. Sc. Benjamin Mensing	TU Braunschweig
Prof. Dr. Andreas Rausch	TU Clausthal
Prof. Dr. Kurt Schneider	Leibniz Universität Hannover

The ruleIT methodology

ABSTRACT

Modern living environments are increasingly supported and guided by software-based systems. IT ecosystems consist of a wide variety of different technical and socio-technical components and subsystems. Many are controlled by software as autonomous agents. However, humans and IT subsystems interact heavily and unfold emergent behaviour that cannot be anticipated or controlled by traditional engineering approaches.

In this document we sketch the fundamental ideas of a software engineering methodology that is tailored for building and maintaining components of a balanced IT ecosystem. This methodology starts at tailor-made techniques for eliciting feedback and requirements in the interconnected environment of an IT ecosystem. Some of the constraints and requirements are transformed into rules.

Those rules guide architectural decisions and model-based software development. In particular, rules enable verification and validation of actual vs. desired behaviour. Since an IT ecosystem is highly dynamic, an equilibrium may always shift or break apart. Runtime verification of behaviour against rules can help to watch for unwanted effects.

This methodology description is an overview rather than a detailed in-depth discussion of all possible development trajectories. We emphasise the importance of involving human stakeholders - and the essential role of rules to create a new software development approach. To illustrate the interaction and combined benefit of all our research aspects, a SmartAirport check-in example is investigated. It is an integral part of the overall Smart Airport example of the IT ecosystem project. Further work in requirements, architecture, and model-based verification will use this overview document for orientation and guidance.

1 INTRODUCTION

Our work is part of a broader enterprise concerned with IT ecosystems, conducted in the NTH school for IT ecosystems [18]. We first characterise an IT ecosystem. The following features have been identified as crucial properties [7]:

- An IT ecosystem is a system of interacting, partly autonomous subsystems. Humans with their properties and wishes may be conceived as parts of IT ecosystems.
- We consider different forms of changes in IT ecosystems. *Adaptivity* refers to shorthand autonomous changes in behaviour, planned a priori or online, as a reaction on the environment. *Modification* is considered in the sense that IT ecosystems are open dynamic systems where configurations change at runtime without a predetermined plan. Beyond these, midterm or longterm evolution of systems is considered, which may be driven by the stakeholder or by system components. In this case even the regulation and control mechanisms may change.
- A central concept for IT ecosystem are guarantees or constraints which are to be kept up even under evolution steps. A notion of equilibrium is to be defined. Regulation and control mechanisms have to be developed.

In the AIM project of the IT ecosystems school, a bottom-up approach is pursued. In an agent based approach, mechanisms for the coordination between autonomous subsystems are developed. In the ruleIT project, we complement bottom-up strategies with top-down elements. The guarantees and constraints play a central role for ruleIT and lead to the concept of rules, as explained in detail in the following sections. Evolution is carefully introduced as development steps conducted by system designers rather than as automated steps inside the system. Thus, evolution is predetermined by stakeholders or system components. Adaptivity is considered on the level of system components. Modification is allowed in terms of online reconfigurations of interactions between components.

We are interested in IT ecosystems that constitute the environment of human actors and users who are considered part of the IT ecosystem. Stakeholders have needs and desires associated with that environment. They may benefit from interconnected information sources. Human stakeholders interact with components of the system and, thus, contribute to its inherent dynamics. Social networks and Web 2.0 can be used for the exchange of relevant user data.

A modern airport can be seen as an IT ecosystem: Hundreds of public displays, flight schedules, and information systems are connected with input from the gates, from baggage handling, and from various other sources. Check-in is a bottleneck for all passengers. Although they may be in a hurry to reach their flights, passengers may need to wait in line. Many check their mobile phones or personal assistants while they wait. In a full-blown IT ecosystem, those personal devices should be fully integrated with the information infrastructure. There is no need to spend the waiting time idle, and there are so many things one could do with an empowered personal device. We want to support software and device developers to create products that fit well into a balanced IT ecosystem.

The mission of ruleIT is to support software development for components, subsystems, and the infrastructure of an IT ecosystem. On the one hand, independent development of components and the flexibility of autonomous behavior should be maintained. On the other hand, ruleIT wants to provide and support mechanisms for controlling the behaviour and emerging effects of an IT ecosystem. ruleIT is determined to develop a methodology for software development based on rules. Requirements of diverse stakeholders are elicited and collected in a way tailored for IT ecosystems. Software is implemented in models, which may be verified and validated against the rules. Most of software

development occurs during "design time", i.e. outside of the running IT ecosystems. However, important phenomena emerge during runtime.

Rules will be used to describe essential concerns and requirements of stakeholders. We conceive a second type of rules to formalise the core behaviour. This type of implementation-oriented rules is supposed to bridge the gap between needs formulated together with stakeholders, and concerns of implementation and verification. The latter need a clear reference. Verification during design time and during runtime tolerates autonomy and flexibility, while rules constrain its behaviour. A rule can detect undesired behaviour independently of its reason. Therefore, elements of an IT ecosystem can be developed by different providers as long as they comply to development rules and methodology described below. Those new elements can enter the system or leave it. The resulting dynamics cannot be anticipated or constructed using traditional software engineering approaches. Software is a core asset in an IT ecosystem, as it controls devices, displays, and various activators.

IT ecosystems challenge software developers in new ways: Customers, users, and participants of an IT ecosystems must be enabled to share their expectations, requirements, and feedback concerning functionality and quality. Many stakeholders are not IT experts. They neither understand how the autonomous components work and interact, nor can they anticipate their emergent dynamics. Along the same lines, system requirements and rules cannot easily be translated into dependencies between system components since system and requirements are in constant flux.

Rules must be derived from global constraints, and elicited from on-going stakeholder interactions with components of the IT ecosystems. Elicitation must take into account the above-mentioned specific situation encountered by stakeholders. Appropriate innovative techniques for requirements elicitation and validation are required. While many requirements are handed down to model-based software development, a selection of demand is presented as rules. High-level rules are close to requirements, whereas a second type of rules is optimised for verification. The gap between those two types of rules is closed by a software design activity. Architecture of components and infrastructure can adapt certain aspects of requirements-level rules and configure architecture and connectivity of subsystems. Thus, system behaviour can adapt to situations perceived by sensors and input devices. Both requirements-oriented and implementation-oriented rules must be considered.

Stakeholders of an IT ecosystem will consider an IT ecosystem useful and reliable only if their key requirements are observed at all times. Otherwise, the IT ecosystem may break apart. The next subsection characterises IT ecosystems in more detail. We then introduce our Smart Airport scenario. An overview of our methodology is provided in Section 3. Sections 4, 5 and 6 explain the core activities of our ruleIT software development methodology: Requirements, model-based development, and architecture. A case study discusses the overall process in the ruleIT methodology.

2 SMART AIRPORT AS AN IT ECOSYSTEM

To demonstrate all research questions of the different projects within the *NTH Focused Research School for IT Ecosystems* a (smart) airport was chosen. Our Smart Airport consists of a lot of different systems other (autonomous) systems or people interact with. Also due to the complexity of interaction, the different systems, and the people as part of the IT ecosystem, a Smart Airport is well suited as exemplary scenario in which all research questions can be demonstrated.

A joint exemplary scenario was worked out in cooperation of all projects (AIM, LocCom, and ruleIT) within the *NTH Focused Research School for IT Ecosystems* [5]. This scenario describes a usual day at a Smart Airport where an IT ecosystem with a lot of interacting IT systems is established. Different passengers (named Anna, Bob and Chris) are accompanied during their way to, over, and away from the airport. They all use a mobile device, called *SmartFolk*, to interact with the IT systems at the Smart Airport. A SmartFolk can be imagined as a device like a PDA. Within the IT ecosystem they represent their owners and act as interfaces to the airports IT ecosystem. During their stay at the airport Anna, Bob and Chris get in touch with the traffic management system, the autonomous transport system, the advertisement system which displays them offers on their SmartFolks, the advanced feedback mechanisms, and many other innovative functionalities of a Smart Airport.

Within the overall airport scenario a small and more detailed part serves us to demonstrate our ruleIT research questions. This mini scenario belongs to the check-in process of an airport (see Figure 1). Usually, in existing airports all passengers have to wait in possibly long queues in front of the check-in counters. Additionally to this arbitrary solution, our Smart Airport provides the possibility to book check-in slots. If the passengers decide to book a check-in slot, they do not have to wait in a queue and get a check-in guarantee. Exemplary, this guarantee assures that the passengers are checked-in half an hour before their flight departs, if they are at the airport at least one hour before their departure. In the case, that all slots within the next, e.g., three hours are assigned and no more passengers are able to book a check-in slot, a further check-in counter may switch into the slot mode. During the whole check-in process, the passengers have the possibility to give feedback about the process using their SmartFolks.

FIGURE 1: CHECK-IN AREA OF A SMART AIRPORT. ON THE LEFT SIDE THE PASSENGERS WAIT IN CLASSICAL QUEUES AND ON THE RIGHT SIDE THE PASSENGERS ARE CALLED ACCORDING THEIR ASSIGNED SLOTS.

Behind the scene of this scenario two systems are important: the *scheduler* and the IT ecosystem *controller*. The scheduler is responsible to assign the slots according to the departure time of the passengers' flights and the passengers' arrival time at the airport. Furthermore, the scheduler could also consider the amount or constitution of passenger groups (e.g., little children or older or disabled people). The task of the IT ecosystem controller is to supervise the whole system and to ensure all defined guarantees like the check-in guarantee described above. In the case a guarantee is violated the IT ecosystem controller is responsible to initiate actions which solve this conflict. These actions could comprise adapting distinct components or reconfiguring the system. Moreover, even development of new components or in some cases modification of guarantees is necessary.

Looking at the scenario description we can explain our view on an IT ecosystem and the properties which characterise it like explained in [7]. First, within an IT ecosystem many people and systems interact in a complex environment among and with each other. This property is obvious: every day an airport is visited by thousands of people passing different areas (waiting area, check-in terminals, passport control, boarding area) and interacting there with many systems. In turn, these systems interact with each other to fulfill their duties and responsibilities and form a system of systems.

In summary, the key properties of an IT ecosystem which are autonomy, dependability, and adaptivity are shown in this scenario. The IT ecosystem at our Smart Airport operates autonomously in the sense described above. To take care for the dependability of the IT ecosystem and keeping the system in a state of equilibrium, the mentioned guarantees were introduced. Considering the behavior of reacting to new environmental conditions, the IT ecosystem at our Smart Airport achieves the passengers needs and shows its potential of adaptivity.

3 THE RULEIT METHODOLOGY

For the ruleIT methodology we assume that we are confronted with a running IT ecosystem which has to be modified according to changing requirements of users or the system evironment. We assume that the system has been built according to our assumptions about essential features of IT ecosystems including rules guaranteeing the proper behaviour of the system, in particular with respect to a notion of equilibrium. We are not considering here the situation of constructing an IT ecosystem from scratch. The scenario we will detail in the following sections is to adapt an IT ecosystem to a new situation.

3.1 Overview

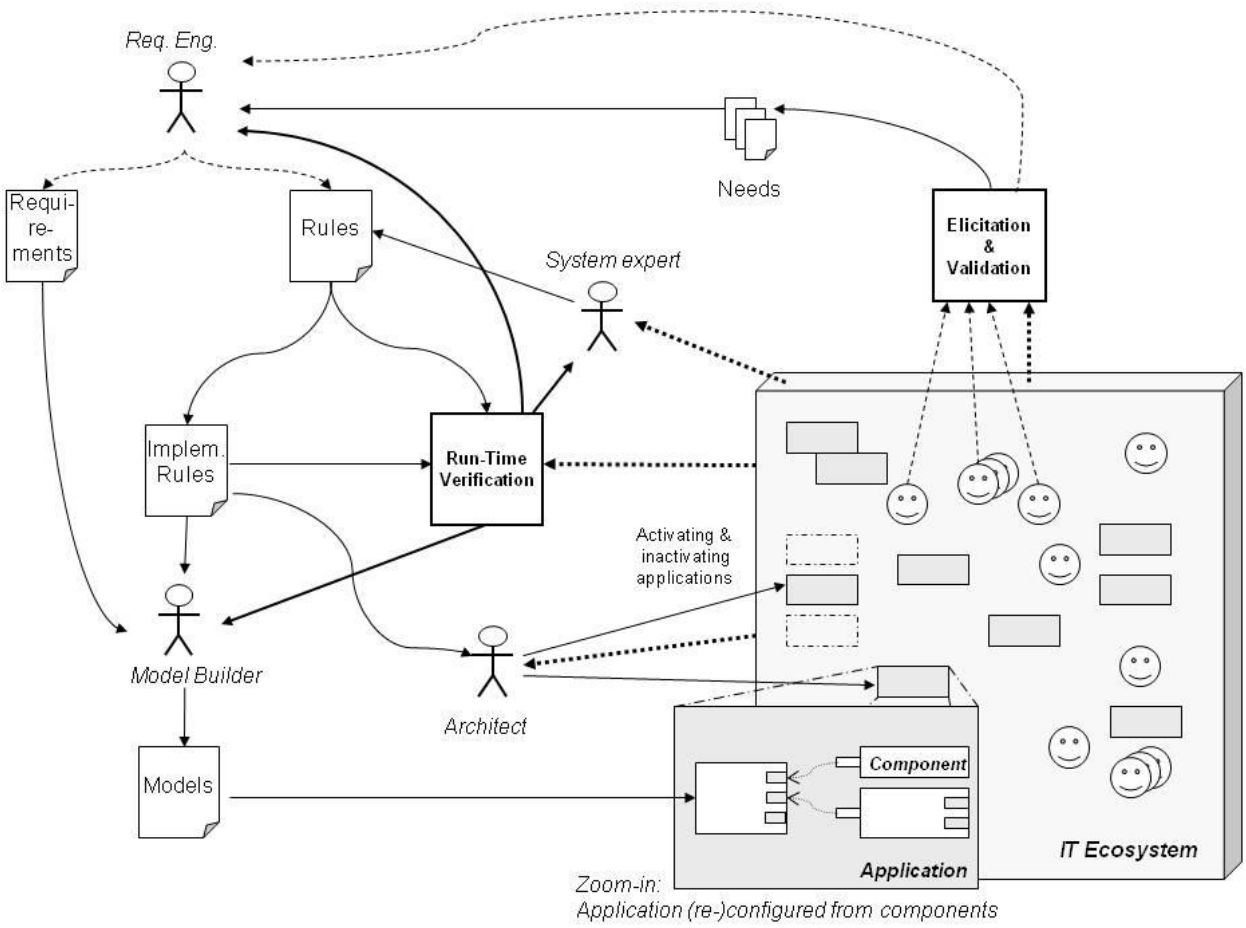

FIGURE 2: OVERVIEW OF THE RULEIT METHODOLOGY

Figure 2 shows an overview of participating roles (stick figures), documented pieces of information (document symbols) and information flows in a software development effort. We use an extended version of the FLOW notation (see [19]). The elements used here are explained in detail below. Figure 3 provides the legend of symbols used in the overview above, and the more detailed version of the methodology (Figure 4) below.

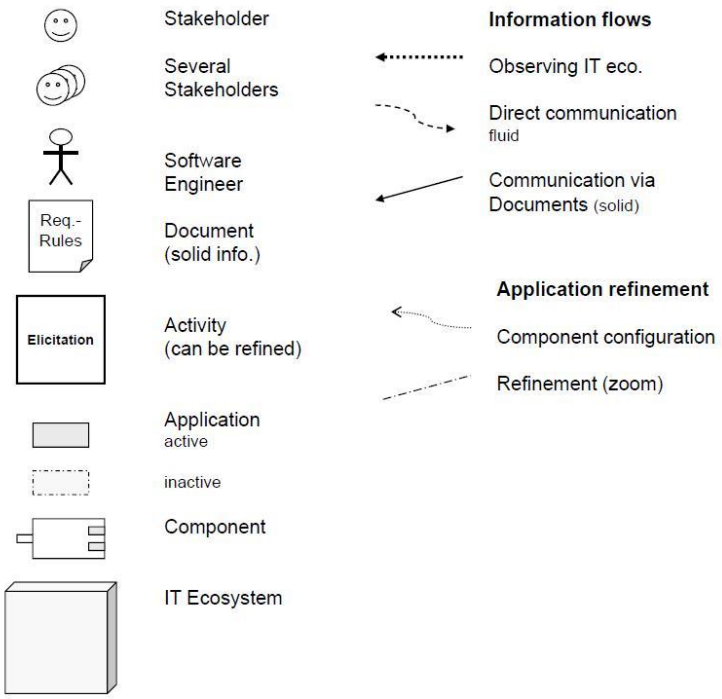

FIGURE 3: LEGEND OF SYMBOLS USED TO DESCRIBE RULEIT METHODOLOGY

Providing software and applications for an IT ecosystem leads to iterations and feedback cycles. These help to keep the system under control. In Figure 2, one might assume an operational IT ecosystem on the right. It consists of applications and subsystems (components etc.), as well as human stakeholders who are also part of the IT ecosystem. As described in the introduction, all elements interact heavily and may change over time. Our approach wants to stimulate input and feedback from stakeholders while they are active in the IT ecosystem. Elicitation is encouraged by providing opportunities for raising needs and turning them into small, but documented pieces of information. This is expressed in Figure 2 by the dashed lines turning into solid documents that can be passed on and analysed by the requirements engineers. An essential task of requirements engineers is the identification of needs that should be turned into "requirements rules", which represent rules on the requirements level. Conventional requirements are documented as such; not all requirements must be transformed into rules. Those traditional requirements are passed on to the developers - in our case, models are developed and used to generate most of the code. In particular (bottom of Figure 2), components are generated from models. Those components need to be configured into applications. During runtime, reconfiguration can become necessary, and entire applications will be enabled (activated) or disabled (inactivated) by the architects. This process is guided by "architecture rules", which are summarized in a box with "implementation rules". Within the ruleIT methodology, we try to automate runtime activities as far as possible.

Requirements rules draw particular attention and must be observed by all subsequent software engineers following the ruleIT methodology. First, they are transformed and detailed into "architecture rules", which guide architectural decisions. These are further refined into "implementation rules". Those rules are sufficiently formal to accommodate design time or runtime verification. Verification is carried out comparing the implementation artifacts against rules. Moreover, models and configurations are compared to rules.

Since the behaviour of an IT ecosystem cannot be predicted, runtime feedback and validation are essential. Therefore, aspects of the "behaviour", including emerging behaviour, are observed. This is indicated by dotted lines from the entire IT ecosystem to the respective roles or activities. For example, runtime verification needs to compare actual (perceived) behaviour with desired behavior encoded in

rules. Validation has the intention to compare real stakeholder needs and desires with actually observed behaviour (dotted arrow to validation indicates observation). Slightly different is the behaviour perceived and commented by stakeholders which flows along the dashed arrows from stakeholders to validation, since it represents information directly communicated by stakeholders. As verification is used to ensure the system's compliance with rules, validation is necessary to make sure that user's requirements are fulfilled.

The ruleIT methodology is supposed to guide domain experts when they provide software for an IT ecosystem. Those experts are represented by the role of "AIM expert". Obviously, other experts from LocCom or simple airport domain experts may also need to be considered: It is their product ideas that will drive software development. They may add or modify requirements and rules depending on their product development strategies and release plans. They are also in charge of resolving conflicts that may be discovered during verification. Keeping an IT ecosystem in balance is difficult on many levels. The contribution of ruleIT is supposed to be a disciplined and innovative approach for software development.

Figure 4 is very similar to Figure 2, but it contains a few more details. Whereas software engineering roles (stick figures) were used in Figure 2 to refer to an activity of that role, Figure 4 makes explicit the core activities carried out during the ruleIT methodology. The core activities were briefly introduced above and are explained in detail in the following chapters. The additional elements make the figure less clear to understand; at the same time, they highlight the essential challenges we are facing in ruleIT.

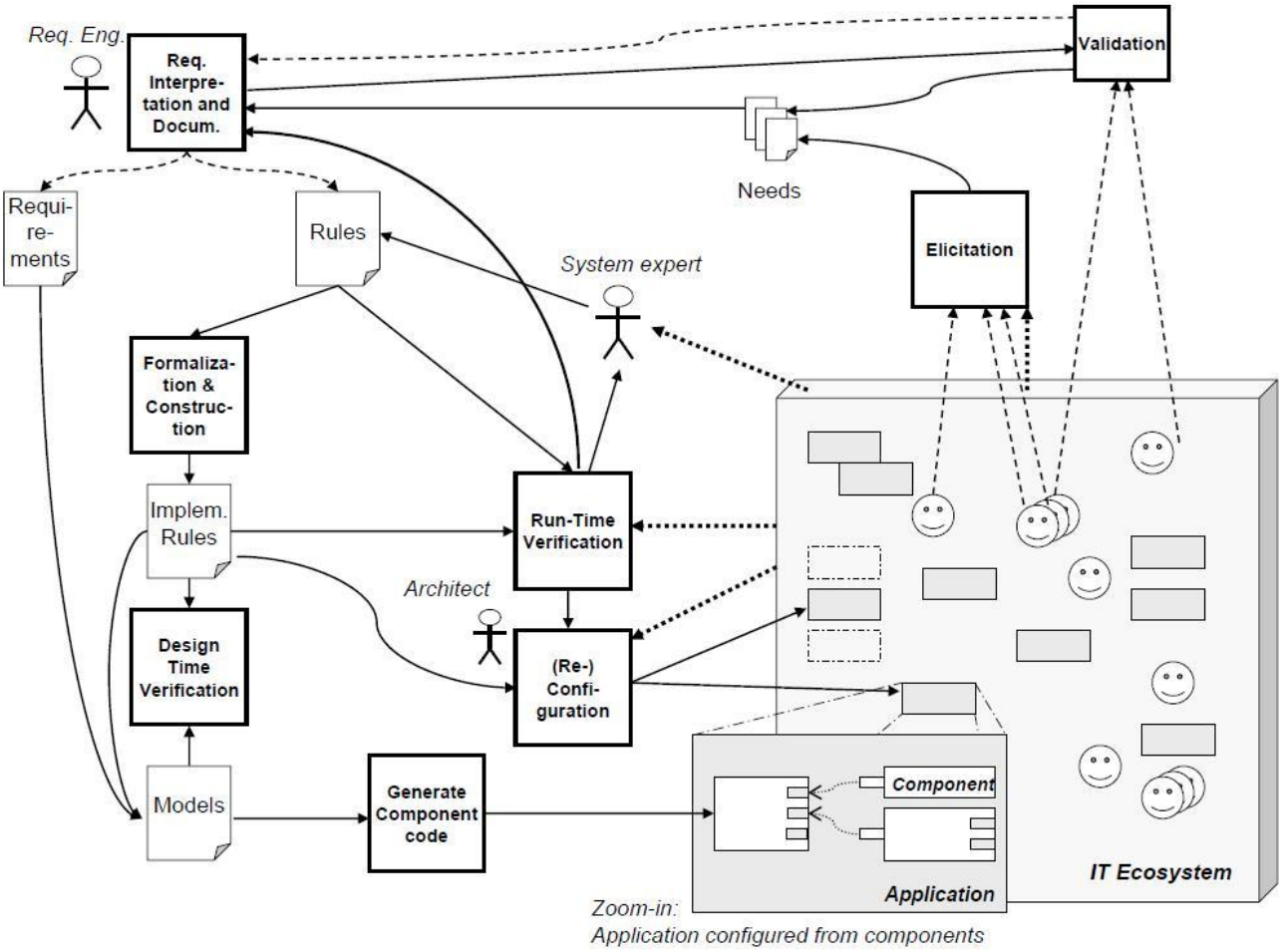

FIGURE 4: DETAILED VIEW ON RULEIT METHODOLOGY INCLUDING ACTIVITIES SQUARE

3.2 Introduction to rules

In a given IT ecosystem, made up of distinct individual systems with complex interaction, unforseen behaviour emerges. To keep the system at equilibrium, some regulatory mechanisms are necessary. These mechanisms have to work in two directions. On one hand, controllability has to be ensured. On the other hand, the system has to be kept useful for its users. Somewhere in the middle, the system resides in a state of balance. Adding too many restrictions enhancing the controllability leads to a system fully controllable, but maybe the usefulness as for the user becomes low. Striving to achieve maximum usefulness, the system possibly cannot be controlled anymore.

To ensure both controllability and usefulness, the IT ecosystem features rules. These rules are of special importance for the IT ecosystem, as rules have a significant influence both on the behaviour of the system and on the usefulness for its users. In ruleIT, we explore how rules have to be shaped and handled to keep the system at balance. Summing this up, rules are essential to control the IT ecosystem.

Rules occur as either requirement rules or implementation rules. Requirement rules are created by the requirements engineer, and represent which functionality should be provided to the user. Precisely, they describe the users' expectations on the implemented component. On the other hand, implementation rules concern the concrete implementation of a component. Moreover, implementation rules serve not only as input for the model builder, but also as input for the software architect and the automatic runtime verification.

As an IT ecosystem is set up, some fundamental rules are formulated to frame fundamental principles of its operation. This initial set has to be provided since a start configuration is needed. Whenever a component for the IT ecosystem is developed, the developer has to take care that it obeys the system's rules. Compliance to the rules is ensured both at design time and at runtime using verification techniques.

Because of the adaptivity of the system, rules can be newly created, edited, or even deleted at runtime. Using elicitation and validation, new needs can be captured from the IT ecosystem. This is, for example, done by interrogating users or by monitoring the system's behaviour (see Section 4).When new needs are identified, respective requirements and requirement rules are formulated. Requirements will be presented as use case diagrams, use case tables, qulity requirements and so forth. In case of existing conflicting rules, the requirements engineer has to solve the conflict by editing the rule. If a rule proves to be improper, it can even be removed. Thus, not only the system is adaptive, but also the set of rules is affected.

3.3 Rule Template

Since rules have to be passed between different participants and processed by the IT ecosystem, a standard layout of a rule is mandatory. To capture all important information, for every single rule an instance of the rule template (Table 1) has to be filled out. The goal of this template is to collect all relevant information that belongs to a single rule, thus it can be passed to and understood by other participants of the development process. Moreover, as a formalisation of the rule is given, it can also be processed electronically. Thus, the rule template becomes human- and machine-readable, which makes it a fundamental part of the ruleIT methodology.

Property	Value
Global Rule / Guarantee	Formalisation of the rule in predicate logic. As expressions like method calls are allowed, these have to be explained in property "Terminology" to allow designers to understand their meaning. A formulation of the rule in a formal logic is especially necessary to proof correctness or compliance of a distinct component to this rule.
Definition	A definition of the rule, written in pseudo-code, is given here. On one hand, this definition is understandable by a human reader, but it is also helpful for the development of components to design them according to the rules. The definition must describe exactly the same rule as declared in "Global Rule".
Hierarchy level	This denotes an unambiguous number, specifying the level where the rule is located on. As an IT ecosystem is a complex system, containing independent components designed by different developers, a hierarchy has to be introduced to solve conflicts between rules. Due to safety aspects, a set of security rules has to be specified on a very high hierarchy level, which makes it hard to change or to vote them down. Also other rules are imaginable that should only be outvoted by a few distinct and important other rules. This technique is a significant contribution to keep the system at equilibrium, since individual systems have to obey rules from all levels above them. A rule for a component on a higher level may also influence components on levels below its own level to facilitate rules containing relations between components on different levels.
Subsystems	In the hierarchical view, the systems referenced here are located on the next-lower level. This enables the developer or the processing system to keep track of the context of the rule. The subsystems referenced here have to be consistent to the "Hierarchy Level", that is if the "Hierarchy Level" is *n*, all subsystems refered to have to be on "Hierarchy Level" (*n+1*).
Dependency on	As some rules are not reasonable without other rules, references to rules (expressed via unique IDs) can be specified here. This makes it easier for the requirements engineer, developers or systems which use the rule template to classify the rule.
Comment	As the properties given above are useful for a software system to understand, they are not suitable for most of the participants in the developing process. Thus, the definition of the rule should be explained in a coherent text here. As a consequence, this row will be considerably larger than the mathematical formulation above, but on the other hand more readable to designers.
Terminology	The wording used in the formulation of this rule has to be explained here. This part of the template is crucial, as the goal of the template to be an exchange format for rules can only be achieved if the definition of the rule can be understood by other participants later. To keep the specification of the rule (both mathematical and verbal) short and clear, additional information about the used vocabulary and what the author meant using a special expression are preserved here. Moreover, expressions like method calls in the pseudo-code or in the logical formulation of the rule have to be explained here.

Assumptions	As most rules are not valid unrestrictedly, the assumptions which have been made when shaping the rule have to be declared here. The environment of a component consists of other adaptive components and humans. This makes it hard to imagine all possible configurations when the rule is formulated, thus some assumptions have to be made, especially concerning the environment. To use these assumptions later in the process, they have to be formulated in a mathematical manner.
Validity / Requirements	This property defines when a rule has to be applied or when it is valid, e.g. if the rule is only applied at a specific point in time. By using this row, special rules can be defined that are only applied in special situations. Moreover, this field enables a combination of different rules, such that e.g. a rule is only applied if another rule failed before.
Obligation	The obligation describes the strictness of this rule, e.g. "hard" or "soft". This becomes necessary if the application of a rule should not be done "binary", which would mean that a rule is either met or violated by a component. In some situations, a more sophisticated technique is required, enabling the specification of an allowed degree of violation, e.g. an interval of values or a number of violations per time.
User group	The user group is the group of people this rule is engaging and applicable for. As some rules are only significant for a special group of users, but inoperative for others, this differentiation can be done by the user group. The user group has to be defined system-wide, such that all systems process this part of a rule in the same manner.
Keywords	Keywords are used to describe the options of this rule, which is essential for context-dependent reconfiguration. Using this set of words, it becomes possible to anticipate which scope the rule has and which other rules are related. Each keyword represents one option to reconfigure the system. Assuming that more than one keyword and analogously more than one option exist, the system presents those options to the user at runtime; Furthermore, the user chooses one option and this option is used as the input for the system (re-)configuration process.
Strategy	To keep the system at equilibrium several strategies are taken into consideration to fulfil the described rule. Those strategies are used as the basis of the control mechanism to ensure the controllability of the system and to keep the system at balance.

TABLE 1: THE RULE TEMPLATE

4 REQUIREMENTS

4.1 Motivation

IT ecosystems comprise a wide variety of sensors, actuators, and subsystems. The users are also part of an IT ecosystem and interact with it by using their SmartFolk. An IT ecosystem and its parts need to be useful to human stakeholders. Hence subsystems evolve and change autonomously in order to meet user and customer requirements.

Traditionally, Requirements Engineering aims at eliciting, handling, and validating requirements. In the realm of IT ecosystems, however, constraints and opportunities differ significantly from a traditional software project environment: There is little chance to interview stakeholders or carry out workshops for elicitation. People, on the other hand, cannot distinguish subsystems easily. Thus, identifying and checking requirements calls for advanced approaches.

4.2 Approach

In ruleIT, we seek to capture a) new requirements and b) restrictions in form of rules. Figure 5 gives an overview of the parts of the ruleIT methodology that are requirements specific. In ruleIT explicit and implicit approaches to requirements elicitation are investigated. After identifying the stakeholders' needs, these have to be interpreted and documented before they are considered to be requirements. This activity is supported by the responsible requirements engineer of the subsystem to be developed. After interpretation and documentation possible solutions for stakeholder requirements must be validated. This part of Requirements Engineering needs special supporting tools because of the special situation in IT ecosystems (i.e. anonymous, unwitting, and large user groups). In the following the steps of the approach concerning the Requirements Engineering part are described in detail:

4.2.1 Elicitation - Derivation of Needs

In IT ecosystems many different groups of stakeholders are present. Each of these groups has its own needs, behavioural characteristics, and preferences. There might be users with a strong desire to have impact on the requirements elicitation, but there also might be users that are not even aware of the subsystems that are part of the IT ecosystem and they interact with.

For each of these user groups, different challenges might be encountered when eliciting requirements for the IT ecosystem. While the technophile might want to deepen in technological details during discussions, the unconscious end user might not only be unaware of the IT ecosystem, but also have no interest at all in helping in requirements elicitation. Also, end users might not even be aware of having any requirements. Nevertheless, to improve the IT ecosystem for its stakeholders, those requirements need to be found out, too.

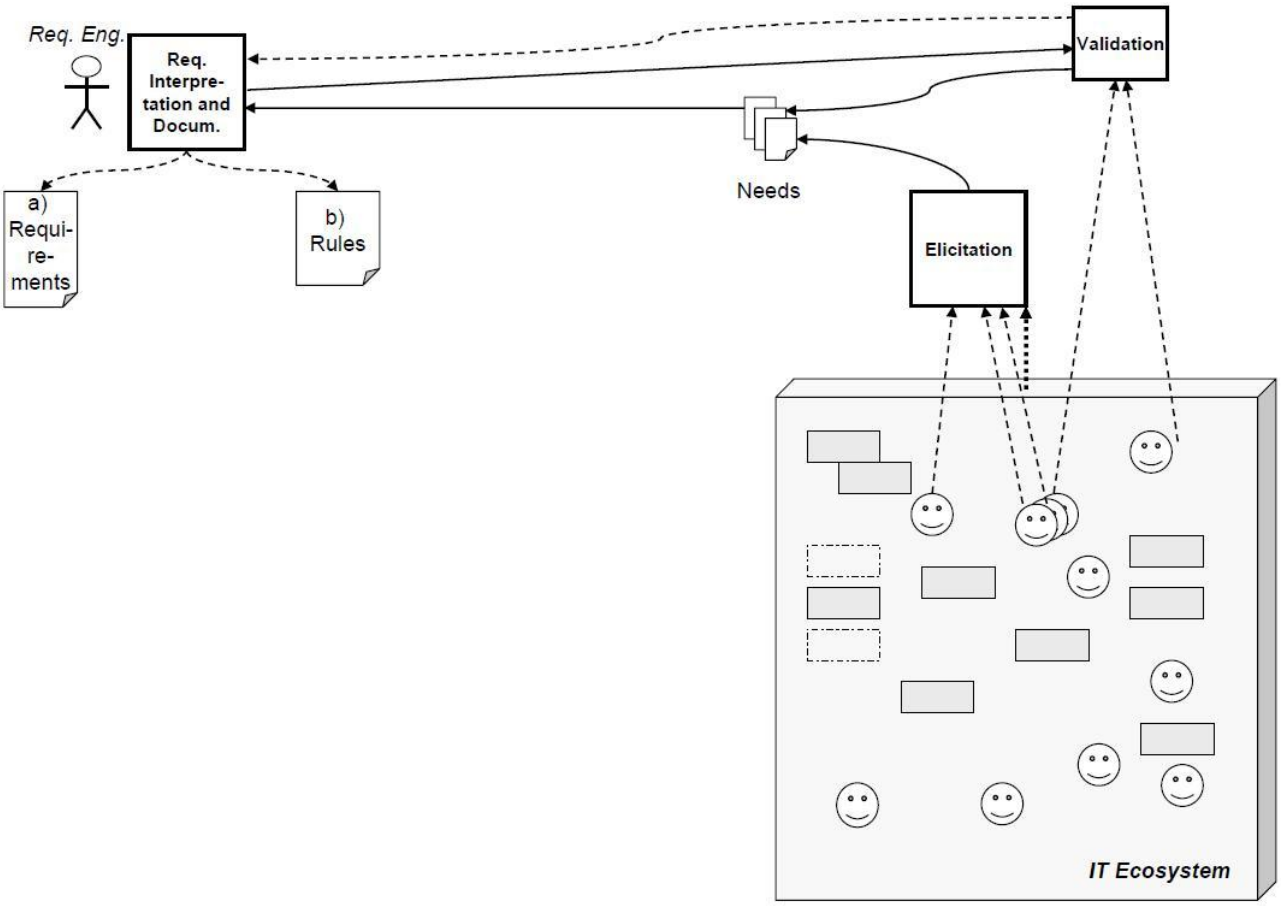

FIGURE 5: THE REQUIREMENTS PART OF THE RULEIT METHODOLOGY

Requirements Engineering methodologies in these environments therefore need to fulfil a set of different requirements themselves in order to be effective. Their cost to stakeholders, in terms of money, time, or anything else, should approach zero, as end users as those described beforehand do not consider themselves to be involved with the system at all. The benefit also must be perceptible for the user otherwise even the technophile will lose interest in contributing his part to the Requirements Engineering.

These requirements for the requirements elicitation lead to a splitting into two parts: the explicit and the implicit requirements elicitation. The explicit part for those stakeholders with interest in the elicitation of requirements and the implicit part either for those without interest in the IT ecosystem or those who have no time to invest in the refinement of the IT ecosystem.

An example for the explicit part is to integrate example questions into the actual usage of the systems' services. Those questions allow users to give quick feedback to features they just used. This could even be part of everyday devices like smartphones or navigation systems. Another approach to make the explicit questioning of users more lightweight could use short videos that show use cases and misuse cases acted out. This bears the potential for creating a lower barrier for interaction with stakeholders. These videos could be produced at low costs, as they would not be required to achieve a professional look (there is an evaluation for the effectiveness of videos for requirements documentation and negotiation in [3]) - their sole purpose would be to enable the involvement of end users for elicitation and validation purposes.

Explicit Requirements Elicitation. Requirements can be raised explicitly by a stakeholder, while using the system. This can occur when a user encounters an error during his work. Requirements can also be raised, when users discover better ways of reaching the same goal. Users may share these new requirements if they expect an improvement of their situation, for example if their reoccurring tasks get

simpler or better. To communicate their requirements or rather things that bother them the explicit requirements elicitation techniques can be used. Such a technique should benefit from the IT ecosystem characteristics. For example, sensors and ubiquitous subsystems (e.g., smartphone, public displays) are used for capturing ad-hoc and spontaneous requests. Such a request must then be contextualised, i.e. assigned to an appropriate context like a device under validation. In our methodology, a smartphone (also referred to as "SmartFolk" in the ruleIT context) with video is used to record a problem or desired functionality in the form of a video, picture or audio in an IT ecosystem.

Stakeholders can participate in improving their environment by mentioning a new (i.e. a desired) use case and recording it on video. Videos can be sent easily. Subsequent analysis steps need to derive rules from such contextualized stakeholder feedback.

In order to give feedback, each stakeholder has to go through the following steps:

1. *Perceive new Needs:* During the interaction with a given subsystem of the IT ecosystem, stakeholders may perceive new needs or missing functionality.
2. *Record new Needs:* In order to communicate these needs, stakeholders need to document them somehow. In the context of ruleIT, stakeholders can use the SmartFolk to do this.
3. *Identify correct Recipient:* As stakeholders are not necessarily able to identify the subsystem they interact with, it is not easy to identify the responsible recipients for their feedback. The SmartFolk enables us to leverage the context to assist in this task.
4. *Submit new Needs:* The new needs must be transmitted to the responsible recipients.

These steps have to be simplified to intensify the (amount of the) stakeholder feedback.

Implicit Requirements Elicitation. In order to complement the explicit elicitation of requirements, the ruleIT methodology leverages implicit mechanisms to derive and infer requirements, change requests, and rules from unexpected user behaviour. Again, IT ecosystem infrastructure is used to observe user behaviour. If a given subsystem is supposed to be extended by new requirements, certain deviations from suggested or supported behaviour can be identified. For example, a person who declines an assigned parking space might have an (implicit) change request for the recommender system. Appropriate analysis mechanisms need to be designed. Statistic, semi-automatic, and interactive techniques are proposed to derive and validate requirements.

After rule deviations have been logged (described in [20]), it is possible to extract new requirements from them or refine those who already exist in the system. Autonomous agents who break a rule are not always interested in actually specifying a new rule. Contrary to engaged users of IT ecosystems who are willing to give feedback, the implicit Requirements Engineering approach tries to track down requirements that have to be extracted from the logged rule deviations. This extraction has to be done manually by the responsible requirements engineer.

Rules that get frequently violated are more likely to point out changed requirements. Since the logging of rule deviations logs the rule as well, all rules can be ordered by their count of logged deviations. To extract new requirements or refine already existing ones, the deviations have to be checked for patterns. In this step, the context in which the deviation occurred is needed.

It is essential to find out and document the reasons why a certain rule is broken in order to create rules that conform to the individual wishes of autonomous agents. Providing a simplistic feedback mechanism for the user to explain their rule deviations is considered for extending the approach. Now the explicit requirements elicitation can help to get stakeholder feedback (described above).

4.2.2 Requirements Interpretation and Documentation

For each subsystem the derived stakeholder needs must be interpreted and documented by the requirements engineer. Based on the implicit requirements elicitation, the rules dedicated to the needs are known. This facilitates the creation of new requirements or the modification of rules. In the case of the requirements based on the explicit requirements elicitation the feedback is send to the responsible subsystem. In some cases the requirements engineer has to identify the correct subsystem, if the feedback was not send correctly. Then in a second step, the responsible rule (respectively requirement) needs to be identified. The requirements engineer has to do this manually.

4.2.3 Validation - Discover the best solution

There are many possible solutions that satisfy the stakeholder needs. First of all new requirements need to be detected. Then alternatives of possible solutions must be identified. These must be discussed with all stakeholders to find out which solution is the best from the stakeholders' point of view. This is done during validation phase. Because of the limited time stakeholder have we need tool support for validation. Videos can be used to visualise the possible solutions in a few seconds. Stakeholder can give their individual convenience. Most of the requirements engineer's work has to be done manually.

The Vision Catcher is a tool that supports the validation based on videos. [11] gives a detailed overview of Vision Catcher.

4.3 Further Work

Subsystem expectations will be formalized and compared to respective stakeholder behaviour. Differences may lead to new requirements. We call this approach "implicit requirements elicitation". This is still an open research problem. Future implementation work is planned in a tool that supports implicit requirements elicitation. We want to support requirements interpretation and documentation of stakeholder needs. An open issue is the interpretation of videos from a requirements perspective.

5 ARCHITECTURE

5.1 Motivation

IT ecosystems are used by plenty of users to fulfil their specific goals. A challenge coming along with an IT ecosystem is the necessity of automatically adapting itself to a current situation of a scenario. Consequently IT ecosystems need to guarantee that their configurations meet the users' needs according to the actual situation. In order to adapt itself accordingly an IT ecosystem has to derive system configuration alternatives, rules and the aforementioned user goals.

While a hierarchical system consists of subsystems, whose interactions are generally predictable and controllable, an IT ecosystem consists of individual systems, whose behaviour and interaction is changing permanently. In general these changes are not planned centrally but rather result of independent processes and decisions of inner or outer effects on the IT ecosystem [12].

Due to the fact that the user has to select the desired subsystem the so called application out of a wide variety of potential applications, the user is overwhelmed with offers and is not able to find applications according to his needs easily. To complicate matters further, a user may expect different applications in different situations. Especially in situations which are new for the user.

On the one hand the user should only catch sight of applications which are relevant in the current situation and compliant to his user profile. SmartFolk providers offer the possibility to search for applications using keywords and other search criteria, but current usage context as well as the user profile is not considered. A navigation application for example could be provided by various different providers, where each application has its own specific characteristics. The result could be a huge list of navigation applications presented to the user. But only a few may be relevant in a specific situation, for example if the user is handicapped and can only walk with help of a walker. In this case a navigation application is needed which avoids stairs during guidance. On the other hand, the service composition has to be adapted during runtime as well because single services of which the application is built may appear or disappear unexpectedly.

According to this the system may determine more than one configuration alternative that comply with the users' goals. The system may activate one of these configurations, but in specific situations it has not the ability to choose the best configuration regarding to the users' goals. This makes a decision by the user indispensable; for example in case of a big disaster the protagonist who is a first aider may either use the navigation application with the alternative to be guided to the next casualty, or he may use the navigation application with the alternative and consequently the appropriate configuration to be guided to the next exit. Hence the system returns those alternatives to the user. The user chooses one configuration and the binding takes place by the system.

In future, more and more applications will be developed by various providers. Those applications within the IT ecosystem will be no stand-alone-applications anymore, but are composed out of multiple independent services. This results in new requirements and approaches for the development of IT ecosystems.

5.2 Approach

The ruleIT methodology introduced above is divided in three research topics; the requirements engineering, modelling and verification and architecture.

In this section we first give an overview of the main components of the architecture of the ruleIT methodology. Second, we present an approach on the architecture level as part of the ruleIT

methodology which enables the automatic context-aware binding of services during runtime to IT ecosystems; so called dynamic adaptive systems [16].

For this purpose, we introduce an approach for the usage-aware application reconfiguration. The result of the 'Modelling and Verification' process is a variety of appropriate services. Those services may appear and disappear during runtime and are the input of the development process on architecture level.

As the services might be available within the IT ecosystem, the architecture approach comprises the runtime aspects. In this chapter we introduce the architecture of an IT ecosystem. First, we depict the architecture, second we explain the architecture in more detail with involving the runtime aspects.

5.2.1 Architecture Overview

As introduced before the user is considered as a part of the IT ecosystem. His decision is involved in the system configuration process; even though the system configured the applications according to the users requirements and evaluates the configuration opportunities. If the system may evaluate more than one configuration, the user has to choose the appropriate configuration during runtime.

The challenges coming along with IT ecosystems are the consideration of those runtime aspects within the development process and as introduced in 3.2 the rules that keep the system in balance.

Figure 6 introduces the conceptual approach on architectural level.

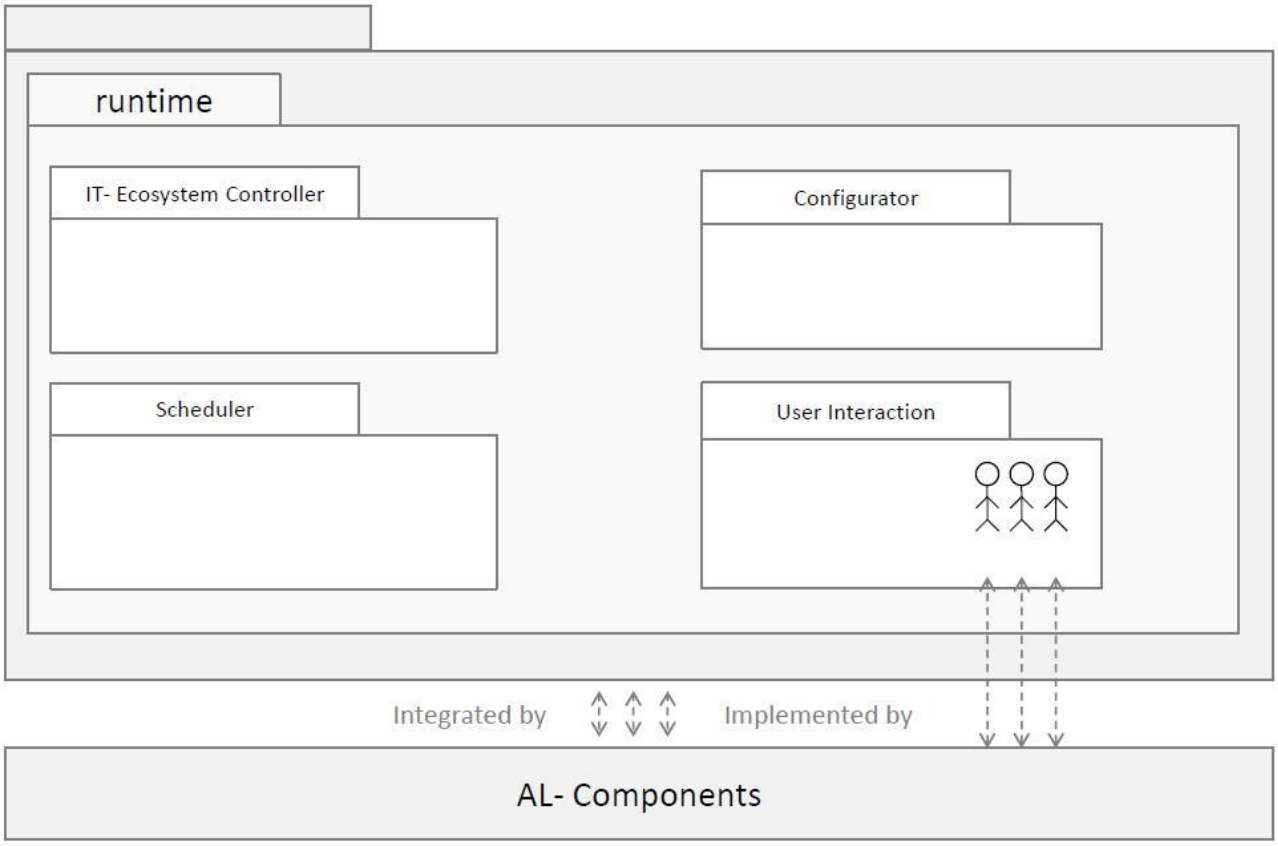

FIGURE 6: ARCHITECTURE: RULEIT METHODOLOGY

The figure above depicts the main components of the ruleIT approach: The IT ecosystem controller, the scheduler, the configurator and the user interaction components are considered during runtime and establish and update the system configuration of the IT ecosystem at runtime.

In our IT ecosystem the IT ecosystem controller builds the centre of the system and controls the guarantees that are derived by the rules out of the ruleIT template.

The scheduler is responsible for the coordination of the services within the application landscape (AL Landscape) and holds appropriate strategies, in our approach the rules, to fulfil the functional aspects of the services.

The IT ecosystem controller verifies the recommendation of the scheduler against the guaranty to observe the rules and consequently to maintain the equilibrium. In case a guaranty is violated, the IT ecosystem controller informs the scheduler. The scheduler chooses another strategy to fulfil the guaranty and returns the evaluated suggestion to the IT ecosystem controller. The IT ecosystem controller verifies this suggestion and informs the scheduler accordingly. This process takes place as long as the strategy chosen by the scheduler match up with the requirements coming along with the guaranty that needs to be fulfilled in the current situation.

As introduced in Chapter 3 'The ruleIT methodology', the user is considered as a part of the IT ecosystem. Consequently, the user interaction component manages the interaction between the IT ecosystem and the user. System configuration of the IT ecosystem are established and updated at runtime ensuring that only dependable system configuration will be established [15]. This takes place by the configurator.

The configurator of an IT ecosystem may determine more than one configuration alternatives of the application that matches with the users' goals and consequently the system returns those alternatives to the user. The user chooses one configuration alternative, the service binding is performed by the system accordingly and the application is returned to the user.

In our application reconfiguration approach services are divided into technical and application services. Those application services 'AL Components' are hold in the application landscape. Furthermore, the ruleIT methodology architecture defines the technical services that are needed for the ruleIT methodology. Those technical services are required for the integration of the application environment and are configured functionally.

5.2.2 Overview of the configuration process

The Figure 7 gives an overview of the configuration process. This process considers the runtime aspects introduced in the previous section. The architecture is divided in four layers. It consists of the Application Selection, Application Repository, Service Composition and the Service Registry layer.

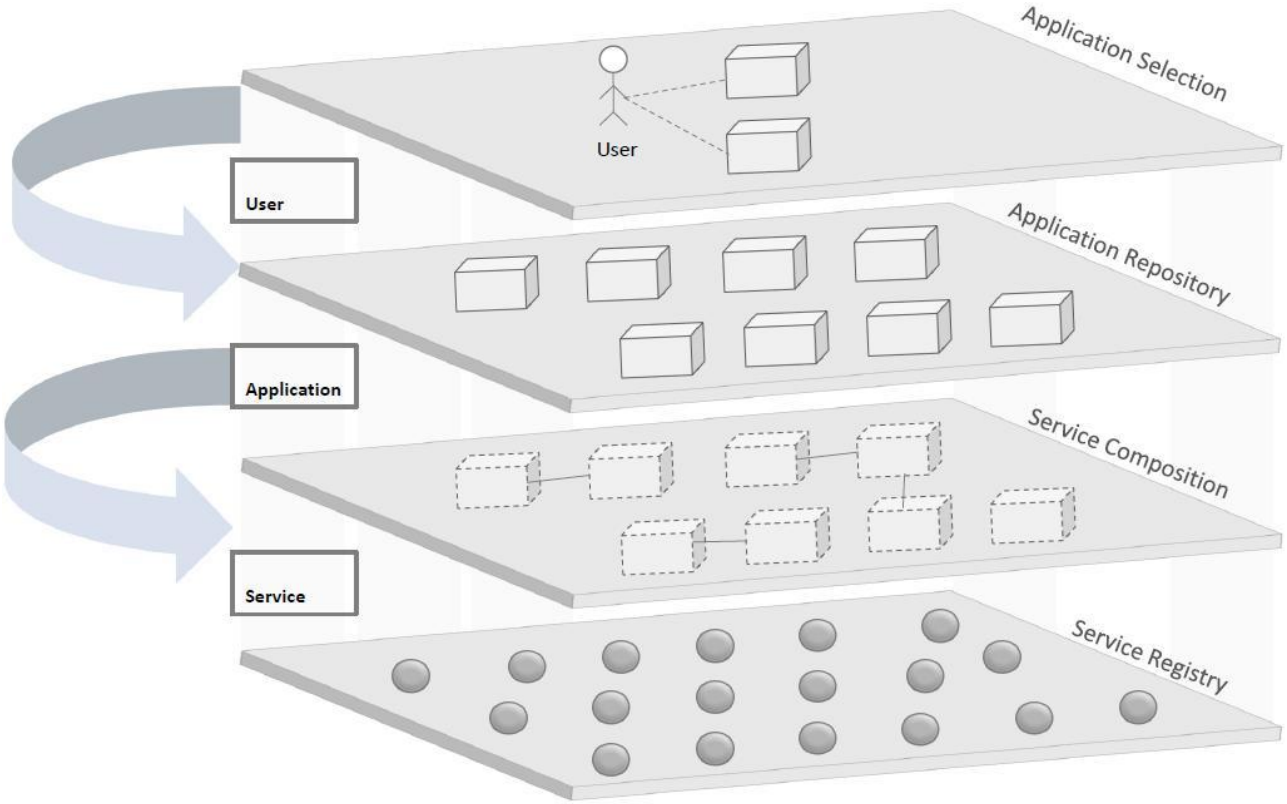

FIGURE 7: OVERVIEW OF THE CONFIGURATION PROCESS

One aspect of our approach is the usage aware application selection. The first layer, the Application Selection, depicts the selection process. In this layer the system determines the appropriate application. To be more precise the system appropriates applications with entirely different functionalities or different variants of applications that matches with the user goals. That means the layer evaluates the application properties with the user profile automatically, to provide the user with the most commensurate application. For this process the system compares the information stored in the ruleIT template, that is passed-through via the Requirement rules (see Figure 2), with the information stored on the users' SmartFolk.

Ideally the system determines one application that matches with the requirements. As the case may be in some situations the system has not the ability to decide which application is best, consequently a user decision is required and the system offers the user the possibility to choose one of the proper applications.

The Application Repository holds all available applications along with a description stored in an application descriptor. Each application is composed of a set of services. The application descriptor specifies possible service configurations and a textual description. As depicted in figure 2, a system architect defines possible service compositions and consequently the basis of the application reconfiguration. The application description holds the configuration options, that are presented to the user and that may possibly composed accordingly, as defined in the Service Composition layer.

The service configuration takes place within the Service Composition layer. According to the description of one selected application in the Application Repository, the services are composed. The service composition for each application is defined in the application descriptor.

In our approach an application is composed of a variety of services. All available service instances that may build up the application are held by the Service Registry within the IT ecosystem. For each service

available via the Service Registry a service descriptor exists. The Service Composition layer catches the required services from the Service Registry; via matching the required services with the registered services.

This general architecture provides the possibility to select and adapt service based applications during runtime.

5.3 Further Work

The results presented in this chapter are more on the conceptual level. In the following we discuss open issues regarding applying the filter to achieve user-specific (semi-) automatic service bindings.

The next step will be to define an optimization approach of the service binding on a more mathematical basis. We plan to use graph transformation to bind services according to the user's goals. First, we concentrate on the approach of the application optimization for one user. The next step will be to enhance the concept considering multiple users.

Within IT ecosystems the complexity becomes a challenge in terms of scalability. We thus need to evaluate the approach regarding this aspect and might enhance the approach by combining the network simplex algorithm with the branch and bound method.

In a first approach we plan to categorize those optimization criteria and thus need to adapt the optimization algorithms accordingly to consider these categories.

This is ongoing work and we have not yet implemented user-specific bindings within our infrastructure for dependable dynamic adaptive systems.

One topic is to define how we can support the user with defining the needs.

6 MODELLING AND VERIFICATION

As systems became more and more complex, keeping systems controllable concerning quality and correctness emerged as an outstanding problem. Text-based programs are hardly sufficient, as the code ultimately becomes barely comprehensible due to lots of lines of code. Furthermore, the traceability is low, thus finding bugs is catchy.

To overcome these problems, model-based software engineering emerged as a possible solution. The model-based software engineering approach features avoiding a lot of problems of manual coding. As a result of tool assistance and graphical notation, this technique is suitable even to guarantee maintainability and changeability for complex interaction in large systems. One of the major advantages of model-based software engineering is the possibility to build executable code automatically from the model. This offers searching of bugs on model level instead of scanning the program code, which makes the developer more efficient. Due to executable models, the bulk of testing can be done on the development platform, where most of the behaviour of the planned system can be monitored.

Because of potential safety properties that must be guaranteed for some systems, also verification is an important aspect of the development process. Whenever a property is essential for the system's functionality or a failure of this function would cause serious aftermath, compliance to the property should be verified. Other important reasons for using verification techniques are enhancement of the system's quality or assuring that the system meets its specifications.

6.1 Introduction

Since IT ecosystems are extremely large and complex software systems, model-based software engineering is ideally suited for their development. The wide range of possible functionality in these systems makes it hard to introduce fast system modifications at runtime unless the comprehensibility of the system is granted. But as changeability and adaptivityare main characteristics of an IT ecosystem, it is fundamental to enable smooth integration of approaches to modify the system's components.

Furthermore, system's variability is not restricted to developer-based modifications, but also rests upon automatic adaption at runtime. Introducing this kind of system-inherent adaptivity, a sophisticated approach is necessary to avoid unintended behaviour. Otherwise, developing a system according to the given requirements would lead to confusing code and most likely to non-compliance of the expected behaviour. To negotiate these problems, elaborate modeling approaches covering adaptivity are necessary. Because established modeling approaches do not scale for systems of this complexity and size, some advanced techniques are presented subsequently.

Moreover, the distinct components of the IT ecosystem are developed independently from one another, maybe from different companies all over the world. Thus, it is crucial to ensure autonomy and controllability of the whole IT ecosystem regardless of giving the developers the freedom to carry out innovative functionality of their own system.

To guarantee autonomy and controllability as mentioned above, IT ecosystems feature rules which have to be met by all components. Especially for ruleIT, these rules are an essential part of an IT ecosystem, because adherence to the given rules has to be enforced to keep the system useful. Moreover, the system's quality can be ensured by rules.

Since rules are partially given at design time, but can also be changed by the system itself or by users at runtime, verification is difficult. Common verification techniques lack of handling of adaptivity and evolution. Furthermore, they don't scale to extremely large systems like an IT ecosystem. To overcome this, techniques as described below are necessary.

6.2 Approach

Given the problems mentioned above, a new software engineering methodology specific for IT ecosystems has to be shaped. The requirements engineering in IT ecosystems, which happens chronologically first, is covered in the 4th chapter. The special architecture of IT ecosystems and the dynamic combination of software components is described in the previous chapter. The part of constructing components which fit into the architecture based on rules and requirements will be explained here.

To classify this part of the ruleIT methodology, the input and output of this step of the development process have to be illustrated. The input of this part mainly consists of use-case diagrams and the standardised rule template. Both are filled with information gathered as mentioned in the previous chapter and passed via standard format. Moreover, information about the system's architecture is needed to identify the respective components.

Use-case diagrams describe what an actor does with a system to reach a goal, without dwelling on exact technical solutions. Because of their high profile as part of the UML and intuitive usage of use-case diagrams, these suit the ruleIT methodology well. Since use-case diagrams do not cover non-functional requirements, their flexibility and expressivity is limited. This gap has to be filled by a standardized format to exchange non-functional requirements, like e.g., timing conditions. A significant aspect here is that both functional requirements in use-cases and additional non-functional requirements are the base to model the system's behaviour.

On the other hand, the ruleIT rule template constitutes the second input for this part of the ruleIT methodology. Rules for the IT ecosystem, which have to be followed by the individual software components, are passed on by the requirements engineering step. The system's compliance with these rules is ensured in this step by distinct verification techniques.

The third part of the input is represented by component and sequence diagrams to show the system's decomposition into separate components and their interaction. Based on these diagrams and the use cases, the behaviour of the distinct components is modeled.

The output of this step is code for the given architecture in a given programming language and additionally the assurance that the code obeys the defined rules and, additionally, a monitor supervising the component. The code itself will be runnable on a CORBA-based architecture [17] without further developer interaction.

To reach the goal of transforming the mentioned input into runnable code and the assurance of compliance to the rules, the software modeller builds a behavior model of the component he develops. Before the code is integrated on the target system, the plain model can be checked regarding correct behaviour according to the given use-cases and non-functional requirements. This can be done by simulation of executable models or model checking techniques. Due to a correct mapping of the model into the code, the most of testing can be done before integrating the code, as already the correctness of the model can be proven. Thus, it is not necessary to perform all tests again on code level if the tools implement the mapping correctly.

After the simulated behaviour of the model meets the specifications, the verification against the rules can be performed. This step ensures that only code that obeys the rules will be included in the IT ecosystem and malfunctions can be eliminated. Especially in systems where the user is dependent on correct functionality, like in IT ecosystems, formal verification is an important aspect. As addressed above, major features of the whole IT ecosystem are adaptivity and evolution.[1] Since components and their

[1] A more detailed view on adaptivity in IT ecosystems is taken in [6]

relationships change over time and even components can be added or removed at runtime, complete static verification of a component becomes impossible. Furthermore, the complexity of these ultra-large-scale software systems complicates verification at design time due to state explosion. Thus, a component to observe and regulate the system at runtime is added, called a runtime monitor. In contrast to static verification at design time, this approach is named runtime verification. In contrast to design time verification techniques, the system's correctness cannot be proven beforehand. Furthermore, only one (the current) execution is observed and compared to the specified behavior.

Following this general description of the modelling and verification step as a part of the ruleIT methodology, different approaches to develop and verify a component for an IT ecosystem are examined. Those approaches are analysed based on some principles which are important for model-based development of a software component for an IT ecosystem. On one hand, the technique has to be applicable for the system's developers, which means it has to be intuitive, clear and universal. On the other hand, notwithstanding all flexibility, the model has to be unambiguous and understandable for other developers having to comprehend and handle them, too. Furthermore, to generate an additional benefit, code generation must be possible directly from the model.

Despite all their differences and let alone the aspect of adaptivity, all of these approaches supply the necessary features to fill out the gap between requirements and code. First, Live Activity Diagrams (LADs) will be presented, later on, Statecharts and Timed Automata are examined. As a fourth approach, Dynamic Automata with Timers and Events (DATEs) will be introduced representing runtime verification. To cover adaptivity, an additional approach which can be combined with some common non-adaptive techniques is presented in the end of this part.

6.2.1 Live Activity Diagrams (LADs)

At first, Live Activity Diagrams are presented. Based on UML Activity Diagrams, Live Activity Diagrams are an add-on, bringing some additional features.

As an behaviour diagram, UML activity diagrams describe the dynamic flow of activities in a model. Activity diagrams are flexible enough to model not only software systems, but also other real-life systems for which behaviour should be specified. Thus, activity diagrams are a widely-known approach for modeling behaviour. Because of the possibility to model a system's behavior hierarchically, even complex and concurrent systems can be modelled.

Due to their high profile, activity diagrams are studied well, moreover, with UML2, their semantics became clearer and closer to a petri-nets-based semantics now. The view on the modelled system is, contrary to other examined approaches, an inter-object view. Thus the focus does not lie on concrete parts of systems, but on activities of the system.

Live Activity Diagrams [13] as an add-on to UML activity diagrams provide hot and cold regions and hot and cold conditions. These constructs called temperature are similar to equivalent constructs from Live Sequence Charts [2]. Hot means, in this context, that an element is mandatory and, for example, a hot action has to be executed completely. Contrarily, a cold element is tentative and does not need to be executed completely.

Especially, this is interesting for local pre- and postconditions, where a hot condition means that violating this condition violates the whole system's requirements specification, too. A cold condition only affects the execution of the current activity, which has to be stopped if the condition is violated.

For modelling and verification in IT ecosystems, this approach is interesting. As mentioned, activity diagrams are very popular and use a graphical representation. Additionally, their upgrading to LADs is also intuitive. Because of the possibility to include hierarchy in the model, even complex systems can be

modelled, so that complexity can be handled. The construct of calls permits reusage of activities. Due to this, a broad group of developers would be familiar with this approach and could work with this formalism.

The verification of LADs could be based on the mentioned hot and cold conditions, where rules would be mapped on distinct local pre- and postconditions. In this way, the system's behaviour can be constrained according to the IT ecosystem's rules. Moreover, violations of these conditions can be detected by validating the model.

Considering adaptivity, activity diagrams seem to be a suitable approach. Different behaviour can be specified in activities which can be called by a controller activity using calls. To adapt the behaviour, only the calls have to be changed dynamically.

Currently, some work is in progress around tool support for live activity diagrams and some tools already exist. First of all, an Eclipse plug-in for simulation of LADs is available. Additional work to control a robot's behavior specified with LADs via Bluetooth was completed in 2009. Ongoing work is engaged with implementing a code generator to build executable Java code from LADs.

A disadvantage of activity diagrams and thus of LADs is their inter-object-view. There is no mandatory construct that declares on which distinct component an activity should be mapped. However, the inter-object-view makes it easier to specify a systems behaviour, as the subdivision into components does not have to be regarded.

6.2.2 Statecharts

After considering LADs, statecharts are examined. The first version, based on finite automata, was proposed by Harel in 1987 [8]. As a part of the UML, statecharts are used to model the dynamic behaviour of a system, similar to the usage of activity diagrams. Contrary to activity diagrams, statecharts use an intra-object view on the system's components and describe the state the system is in, not which activity it actually performs. Moreover, the semantics is semi-formal.

Helpful constructs to model even complex and large systems are abstraction, orthogonality and basic support for time. Abstraction is introduced through usage of hierarchical states. This provides the possibility to abstract from complex behaviour and also allows reusage of parts of the system. Orthogonality is important to model concurrent systems, which is mandatory for IT ecosystems. For this purpose, statecharts provide parallel constructs to model parallel procedures. As time is also an important aspect in many systems, statecharts contains some basic support for time, namely timers. These timers allow triggering a state change after a distinct period or at a distinct date, for example.

Subsuming these features, statecharts are well suited for modelling in IT ecosystems, their scope is very similar to that of activity diagrams. They are, like activity diagrams, relatively easy-to-use and intuitive, primarily because of their graphical representation. Compared to activity diagrams, statecharts are not that universal as activity diagrams are, and even their view on the system is different because of the intra-object view. An obvious advantage is, thus, an easier combination of static and dynamic modelling: While the structure of the system can be modelled using a class diagram, one statechart defines the behaviour of exactly one class. Regarding activity diagrams, this approach is slightly more low-level because of the existence of concrete classes.

For the development of systems using statecharts, some mature tools are available, for example Rhapsody or MATLAB/Simulink. As these tools include a graphical editor, the system's behaviour can be specified and even simulated to test the model before code integration. To generate code from statecharts, some tools are already available, moreover, additional promising approaches have been presented.

Considering the verification of statechart models, model checking tools like for example SPIN have been proposed. Those tools require a formula in temporal logic as input to check a given model. A disadvantage is the relatively high expert knowledge needed to form these formula from rules.

To include adaptivity in statecharts, the hierarchical composition could be used. One approach could be to model the different possible behaviours as various statecharts. These various statecharts would then be encapsulated in distinct states of a higher-level statechart. This higher-level statechart represents a controller component that switches between different behaviour modes by switching into another high-level-state. Thus, the behaviour of the whole system is changed.

6.2.3 Timed Automata

Another approach to model a system's dynamic behaviour are timed automata. Like statecharts, timed automata are also based on finite state machines and show some other similarities to statecharts. Timed automata were first proposed in [1] as extension of clocks with continuous time to finite automata, invariants and the possibility to form networks of automata.

The extension with continuous time enables a more realistic model of real-world processes, but makes verification difficult on the other hand. To model a component for an IT ecosystem, probably using discrete time may be sufficient for simplification firstly, but continuous time is required for realistic modeling of all properties. Using invariants adds a feature to include rules in the model: By specifying invariants, it is possible to declare rules the system has to obey in distinct states. A helpful feature for a more complex system is the possibility to combine several automata communicating through channels. Using this kind of communication, both intra-object and inter-object communication can be implemented.

Verification of timed automata against temporal logic is possible and has some similarities to verification of statecharts. For example, the UPPAAL-Tool contains a front end to model a timed automaton and verify it against a special subset of temporal logics [14]. Due to the explosion of state space, verification becomes slow or even impossible for large systems with a lot of states and clocks.

Similar to statecharts, also timed automata have a graphical representation which makes them very demonstrative. A disadvantage is, comparing them with statecharts, obviously the lack of constructs for abstraction like hierarchical states. Moreover, orthogonality inside one automaton is not possible and has to be modelled using several communicating automata. Thus, modelling a complete software system is hard and confusing, which means that there are hardly any advantages compared to a textual programming language. Due to the lack of hierarchical constructs, including adaptivity becomes more complicated as in other approaches.

6.2.4 Dynamic Automata with Timers and Events (DATEs)

The last approach considered here pursues a different strategy. As mentioned above, not all properties can be verified at design time, mainly because of adaptivity and evolution. A possible solution is to add a runtime monitor to the system, such that the system's code runs as usual, while an independent monitor runs parallel to observe the system's behaviour and intervents if necessary.

One concrete approach for runtime monitoring was proposed by Colombo et al. with LARVA [4]. The idea of this approach is to combine formal verification and program execution. The system is implemented using code generation or manual coding, it's intended behaviour is specified using a special kind of automata, called a DATE, which determines the runtime monitor.

DATEs are automata based on timed automata, but with additional functionality for clocks. Moreover, "bad states" can be used to declare undesirable behaviour. Thus, the underlying system can be monitored and even influenced, for example when a bad state is reached, which constitutes the

verification. The decision whether a state is bad or good is based on rules and requirements. A DATE can be build using a graphical representation, a textual language or an LTL formula. The interventions on the underlying system are given as simple program code.

The LARVA tool, which is used to perform runtime verification of a system against DATEs, uses aspect-oriented programming techniques to monitor and influence the underlying systems. Because LARVA uses AspectJ as an aspect-oriented extension of Java, both the underlying system and the intervention code in the DATE have to be written in Java.

The main advantage of this approach is that the underlying system can be implemented as usual and does not need to care about any kind of further verification. The conditions that should be verified, derived from the rules, are specified at one central place: the runtime monitor. This offers two possibilities for productive use: The runtime may be removed after the test stage or retain in productive use.

Contrary to other techniques, not the correctness of the model, but the correctness of the implementation is checked directly. The system is observed inside its realistic application environment. This is a big advantage compared to tests, which do not cover all aspects arising in the productive use.

Compared to the approaches presented above, this approach suits best to cover adaptivity aspects as changing the underlying system does not necessarily affects the runtime monitor. Moreover, less expert knowledge is needed compared to most other verification techniques, because of the specification of the intended behaviour using a graphical formalism.

6.3 Further work

Subsuming the results presented above, a methodology for model-based software development and verification of IT ecosystems has to deal with adaptivity. Thus, a combination of static verification, e.g. model checking, and runtime verification seems reasonable.

As a next step inside the IT ecosystems project, a combination of static and runtime verification will be chosen and further studied. A main aspect is the possibility to cover adaptivity with the chosen approach. In this context, the check-in scenario should be implemented using this approach. This means, in detail, modelling the check-in based on use-case diagrams and the rule template, and generating executable code from the model. Certainly, verification will be performed.

Moreover, some more approaches for adaptive modelling should be evaluated, especially regarding their suitability for software engineering of IT ecosystems.

To make statements about timing behavior, only adherence to timing constraints in rules will be included, the ability of handling real-time will not be considered.

REFERENCES

[1] R. Alur and D. Dill. A theory of timed automata. *Theoretical computer science*, 126(2):183-235, 1994.

[2] M. Brill, W. Damm, J. Klose, B. Westphal, and H. Wittke. Live Sequence Charts

[3] O. Brill, K. Schneider, and E. Knauss. Videos vs. Use Cases: Can Videos Capture More Requirements Under Time Pressure? In *Accepted for REFSQ 2010, Essen, Germany*, Essen, Germany, 2010.

[4] C. Colombo, G. Pace, and G. Schneider. Dynamic event-based runtime monitoring of real-time and contextual properties. *FMICS08*, pages 135-149, 2009.

[5] C. Deiters, M. Köster, S. Lange, S. Lützel, B. Mokbel, C. Mumme, and D. Niebuhr (Eds.). DemSy - A Scenario for an Integrated Demonstrator in a SmartCity. Technical report, NTH-School für IT-Ökosysteme, 2009.

[6] U. Goltz, N. Khakpour, C. Knieke, and L. Märtin. Behavioral Modeling of IT Ecosystems. Technical report, NTH-School für IT-Ökosysteme, 2010.

[7] U. Goltz, J. Müller, C. Müller-Schloer, and A. Rausch. Was ist ein IT-Ökosystem? Technical report, NTH-School für IT-Ökosysteme, 2009.

[8] D. Harel. Statecharts: A visual formalism for complex systems. *Science of computer programming*, 8(3):231-274, 1987.

[9] S. Herold, H. Klus, D. Niebuhr, and A. Rausch. Engineering of IT ecosystems: design of ultra-large-scale software-intensive systems. In *In Proceedings of the 2nd international workshop on Ultra-large-scale software-intensive systems, 2008, Leipzig, Germany*, 2008. ACM, pages 49-52, 2008.

[10] IBM Software. Rational rhapsody, 2010. *http://www.ibm.com/software/awdtools/rhapsody*

[11] I. Kitzmann. Konzept und Implementierung eines Werkzeugs für multimediale Anforderungserhebung und -validierung. Master's thesis, Leibniz Universität Hannover, 2009.

[12] H. Klus, D. Niebuhr, and A. Rausch. A Component Model for Dynamic Adaptive Systems. *Foundations of Software Engineering*, pages 21-28, 2007.

[13] C. Knieke, M. Huhn, and M. Lochau. Executable Requirements Specification:

Formal Semantics of Live Activity Diagrams. In *Theoretical Aspects of Software Engineering, 2008. TASE'08. 2nd IFIP/IEEE International Symposium on Theoretical Aspects of Software Engineering*, pages 109-112, 2008.

[14] K. Larsen, P. Pettersson, and W. Yi. UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer (STTT)*, 1(1):134-152, 1997.

[15] D. Niebuhr. Dependable Dynamic Adaptive Systems - Approach, Model, and Infrastructure. PhD thesis, Technische Universität Clausthal, 2010.

[16] D. Niebuhr, H. Klus, M. Anastasopoulos, J. Koch, O. Weiß, and A. Rausch. Daisi - Dynamic Adaptive system infrastructure. Technical report.

[17] Object Management Group, Inc. OMG's CORBA Website, 2010. *http://www.corba.org/.*

[18] A. Rausch. NTH Focused Research School for IT Ecosystems, 2010. *http://www.it-oekosysteme.org/.*

[19] K. Schneider, K. Stapel, and E. Knauss. Beyond Documents: Visualizing Informal Communication. In *Proceedings of Third International Workshop on Requirements Engineering Visualization (REV 08)*, Barcelona, Spain, 9 2008.

[20] L. Singer, O. Brill, S. Meyer, and K. Schneider. Leveraging Rule Deviations in IT Ecosystems for Implicit Requirements Elicitation. In *Second International Workshop on Managing Requirements Knowledge (MaRK'09) at RE'09*.